

SOME GRAPH PROBLEMS IN Coq

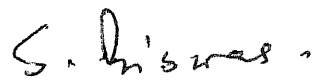
A Thesis Submitted
in Partial Fulfilment of the Requirements
for the Degree of
Master of Technology

by
Manoj.V.K.

to, the
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY
May, 1992

CERTIFICATE

It is certified that the work contained in the thesis titled "SOME GRAPH PROBLEMS IN Coq", by "Manoj.V.K." has been carried out under my supervision and this work has not been submitted elsewhere for a degree.


(S.Biswas)

Dept. of Computer Science
and Engg.

I.I.T. Kanpur

April, 1992

03 JUN 1992

CENTRAL LIBRARY
U.S. AIR FORCE

Acc. No. **A113550**

ABSTRACT

We present in this thesis the formal development in Coq of the depth first search and strongly connected components algorithms. Coq is an implementation of the Calculus of Inductive Constructions under development at INRIA and ENS in France. The Coq system allows the extraction of programs from constructive proofs of higher order logic assertions.

ACKNOWLEDGEMENT

I am extremely grateful to my guide, Dr. S Biswas for his help throughout the course of the thesis. Special thanks to Saya and Mohalik for their help in the thesis report preparation.

Manoj.V.K.

Table of Contents

| | |
|----------------------------------------------------------|----|
| 1. Introduction..... | 1 |
| 1.1 Key ideas behind Coq..... | 1 |
| 1.2 Outline of Thesis..... | 10 |
| 2. Program Development..... | 11 |
| 2.1 Basic Definitions and Lemmas..... | 11 |
| 2.2 Development of Depth First Search..... | 16 |
| 2.3 Depth First Search for Strongly Connected Components | 23 |
| 3. Program Extraction..... | 31 |
| 4. Conclusion..... | 33 |
| Appendix 1..... | 35 |
| 1. The Tactics Theorem Prover..... | 36 |
| 2. Inductive Definitions..... | 37 |
| 3. A Proof with Tactics Theorem Prover..... | 40 |
| Appendix 2..... | 43 |
| References..... | 58 |

CHAPTER 1

INTRODUCTION

In this thesis we present the formal development in Coq of the depth first search algorithm and its application for finding strongly connected components. As the algorithms are well known[1], we devote this introductory chapter to the motivational aspects behind work such as ours and to some key ideas on which the Coq system is based.

Formal programming methods are concerned with the application of logic in the specification, synthesis and verification of programs. The basic motivation for formal methods is that the strategy of 'test a few cases and extrapolate' is inadequate for the total elimination of program errors. Formal methods allow us synthesize programs that provably meet their specification. Application of constructive logic to program development is a major research area in formal methods.

Coq is an implementation of the Calculus of Inductive Constructions under development at INRIA in Rocquencourt and at ENS in Lyon. The Coq system has three main features. The first main feature is its logical language. This language is an extension of the Calculus of Constructions with primitive inductive definitions. The next feature is its proof assistant for higher order logic which allows the development of verified mathematical proofs. The final and the most important feature, from a formal methods view point, is the program extractor which extracts programs from the constructive contents of proofs. From a constructive proof of the statement

$(\text{forall } x).((P\ x) \rightarrow (\text{exists } y).(Q\ x\ y))$

where $(P\ x)$ and $(Q\ x\ y)$ are of type `Prop`, the program extractor extracts a function f such that

$(\text{forall } x)(P\ x) \rightarrow (Q\ x\ (f\ x))$ is satisfied.

Hence to develop a program in the Coq system, we prove, using the proof assistant, an existential statement relating the output value to the input arguments.

1.1 Key Ideas Behind Coq

In this section we briefly review the development of constructive logic, type theory, Curry-Howard Isomorphism and the various extensions of simply typed lambda calculus.

1.1.1 Constructive logics and Curry-Howard Isomorphism

The initial impetus for the development of constructive logic was provided by Brouwer's intuitionistic approach to mathematics. In Brouwer's view, mathematical reasoning could only be safe (free of paradoxes) if based on finitistic proofs (proofs that does not require an infinite verification). Thus the fundamental intuitionistic assumption about proofs is the decidability of $(c\ \text{proves}\ C)$ for proofs c and statements C .

An intuitionistic interpretation of logic based on modeling proofs of assertion was developed by Heyting in 1930. An informal description of Heyting's semantics is given below[5].

- a. For atomic sentences, we assume that we know intrinsically what a proof is.

- b. A proof of $A \wedge B$ is a pair (p, q) consisting of a proof p of A and a proof q of B .
- c. A proof of $A \vee B$ is a pair (i, p) with
 - $i=0$, and p is a proof of A
 - $i=1$, and p is a proof of B
- d. A proof of $A \rightarrow B$ is a function which transforms a proof of A into a proof of B .
- e. In general, the negation $\sim A$ is treated as $A \rightarrow \#$ where $\#$ is a sentence with no possible proof.
- f. A proof of $(\text{forall } x).(A \ x)$ is a function which maps each object c of the domain of definition into a proof of $(A \ c)$.
- f. A proof of $(\text{exists } x).(A \ x)$ is an object c of the domain of definition and a proof of $(A \ c)$.

A consequence of this proof interpretation is that $A \wedge \sim A$ is not provable, since it is not possible in general to find either a finite proof of A or a finite proof of $\sim A$.

Heyting's semantics provide the basis for Curry-Howard Isomorphism. The main motivation for Curry-Howard Isomorphism is the development of a language of proofs and assertions suitable for the above proof interpretation of logic. Church in 1932 had modeled proofs and assertions of classical logic using lambda calculus. This formalization of logic was proved inconsistent since a version of Russell's paradox created a contradiction[6].

In 1908, Russell identified in the lack of predicativity the essence of paradoxes in mathematical foundations. For the resolution of paradoxes, Russell introduced type theory by postulating a stratification of the mathematical universe.

The two main assumptions of Russell's type theory are

- a. Functions differ from individuals. In particular functions of functions, etc result in a stratification of functions. A key result of this assumption is that a function cannot be applied to itself. An example by Stoy for the paradox that results from self application of functions is given below.

let $f = [\lambda y]. \text{ If } (y\ y) = a \text{ then } b \text{ else } a$. Now the attempt to evaluate $(f\ f)$ results in the contradiction

$\text{If } (f\ f) = a \text{ then } b \text{ else } a$.

- b. A description of a level(n) object is a level($n+1$) object. The paradox that results from allowing descriptions to talk about themselves is nicely illustrated by "the smallest integer whose description requires more than ten words"[8]. Since for Russell, the range of significance of a propositional function is a type, this assumption results in a hierarchy of higher order types.

Based on assumption (a) of Russell's theory of types, Church in 1940 introduced the first version of explicitly typed lambda calculus, the simply typed lambda calculus. This explicitly typed lambda calculus can be extended in many ways. Now the essence of Curry-Howard isomorphism is that the simply typed lambda calculus

and many of its extensions provide the right medium to formalize the proof interpretation for various constructive logics, if terms are seen as proofs and types of terms are seen as the assertion which they prove.

We now briefly illustrate the Curry-Howard Isomorphism between propositions of minimal propositional logic and types of simply typed lambda calculus[4].

A type environment Q is a list $x_1:A_1, \dots, x_n:A_n$ of type declarations such that x_i 's are distinct identifiers. Now the derivation rule $(I-)$ for simply typed lambda calculus can be formalized by

- a. $Q \vdash x:A$ if the type environment Q assigns the type A to identifier x .
- b. If $Q, x:A$ is a type environment then
 $Q \vdash [x].M : A \rightarrow B$ if $Q, x:A \vdash M:B$
- c. If $Q \vdash M : A \rightarrow B$ and $Q \vdash N : A$ then $(M N) : B$.

The set F of formulas of minimal propositional logic is the closure of a base set B of propositions under implication. The entailment relation (\vdash) between subsets and elements of F is axiomatized by

- a. $H \vdash A$ if A belongs to H , where H is a subset of F .
- b. $H \vdash A \rightarrow B$ if $(H \cup \{A\}) \vdash B$
- c. If $H \vdash A \rightarrow B$ and $H \vdash A$ then $H \vdash B$.

What we need now is a notation to code the proof steps required to

prove $H \vdash A$. For this purpose we identify propositions with types. Let Q be the type environment obtained by distinctly marking the hypotheses in H . Now if Q marks A by the identifier x , then x is a proof of A . If M is a proof of B in the environment $Q, x:A$ then $[x].M$ is a proof of $A \rightarrow B$ in the environment Q . Finally, if in the environment Q , M is a proof of $A \rightarrow B$ and N is a proof of A , then (MN) is a proof of B in Q . Thus we can easily code the proof steps corresponding to the three entailment inference rules using simply typed lambda calculus. Hence by induction if $Q \vdash M:A$ then the lambda term M formalizes the proof steps required to prove $H \vdash A$.

1.1.2 Classification of Explicitly Typed Systems

Barendregt[3] has proposed a classification of the extensions of simply typed lambda calculus based on the four basic functional dependencies : terms dependent on terms, terms dependent on types, types dependent on types and types dependent on terms. The dependency 'terms dependent on terms' characterizes the functions in simply typed calculus. Basic idea behind this characterization is that if f is a function in simply typed lambda calculus then $(f \ x)$ is a term depending on the term x .

In simply typed lambda calculus, to express essentially identical functions on different types, different functions must be written. Thus $[x:A].x$ of type $A \rightarrow A$ is an identity function for type A while $[x:B].x$ of type $B \rightarrow B$ is an identity function for type B . A solution to this redundancy is to abstract types over terms to get polymorphic functions. Thus $[A][x:A]x$ is a function that takes as input a type B and returns the identity function $[x:B]x$ for type B .

We can characterize polymorphic functions by the dependency 'terms dependent on types' since for a polymorphic function f , $(f A)$ is a term that depends on type A . Now the question is what is the type of a polymorphic function. Since a polymorphic function takes as input a type and returns a term, we need a notation other than \rightarrow to indicate its type. We use λ to bind a type variable in the type of a polymorphic function. Thus $\lambda A.A \rightarrow A$, read as (forall A) $A \rightarrow A$, is the type of a polymorphic function. Extension of simply typed lambda calculus with terms dependent on types gives us the polymorphic lambda calculus (λ).

In polymorphic lambda calculus (λ), we can represent the type of a list with elements of type A by $\lambda X.X \rightarrow (A \rightarrow X \rightarrow X) \rightarrow X$. However to represent the type of a list with elements of type B , we have to repeat the above definition substituting A by B . To remove this redundancy we extend the polymorphic lambda calculus (λ) with abstraction of types over types. Thus the function $\lambda A.\lambda X.X \rightarrow (A \rightarrow X \rightarrow X) \rightarrow X$ takes as input a type B and returns the type of a list elements of type B . Hence we can characterize this abstraction of types over types by the dependency 'types dependent on types' since the abstraction results in a function f such that $(f A)$ is a type that depends on the type A .

To keep these 'types dependent on types' free of paradoxes we need a notion of type of type expressions. We call the type of types kinds. The constant kind $*$ is the kind of types of terms. Thus the type of a type expression that takes as input a type of type $*$ and returns a type of type $*$ is $* \rightarrow *$, the type of a type expression

that takes as input a type of type $* \rightarrow *$ and returns a type of type $*$ is $(* \rightarrow *) \rightarrow *$, and so on. This results in a hierarchy of higher order types given by

$K = * \mid K \rightarrow K$. Thus the type $A \rightarrow A$ of the identity function for type A is of type $*$. Similarly the type $(A)A \rightarrow A$, which can now be more precisely represented as $(A:*)A \rightarrow A$, is the type of the polymorphic identity function, and hence is of type $*$. We can have the even more polymorphic identity function,

$[A:* \rightarrow *][B: *][x: (A \rightarrow B)].x$.

This term of type $*$, takes as inputs a function from type $*$ to type $*$ and a type B and returns an identity function for type $(A \rightarrow B)$.

For eg, given as inputs the list type constructor

$[A](X)X \rightarrow (A \rightarrow X \rightarrow X) \rightarrow X$ and a type B of type $*$,

the even more polymorphic function returns the identity function for lists with elements of type B . This example also illustrates the abstraction of a type A of kind $* \rightarrow *$ in a type dependent lambda term of type $*$. Similarly, it is also possible to abstract types of any kind in a lambda term.

The critical difference between $[A: *]A \rightarrow A$ and $(A: *)A \rightarrow A$ is that $[A: *]A \rightarrow A$ is a type function and has no term of that type, while $(A: *)A \rightarrow A$ is the type of the polymorphic identity function.

The lambda calculus obtained by extending the polymorphic lambda calculus (F) with 'types dependent on types' is called F_w .

For a Curry-Howard Isomorphism suitable for the proof interpretation of predicate calculus, we require the notion of types dependent on terms. For this we extend the notion of kinds such that if A is a type and k is a kind then $A \rightarrow k$ is a kind.

In particular, $A \rightarrow *$ is a kind. Thus for eg, if $P:A \rightarrow *$ and $x : A$ then $(P\ x):*$ is a type that depends on a term (x) . Now by Curry-Howard Isomorphism a term f of type $(x:A)(P\ x)$ can be seen as a proof of $(\text{forall } x).(P\ x)$ since for any x , $(f\ x)$ is a proof of $(P\ x)$. Similarly a term g of type $(x:A)(P\ x) \rightarrow B$ can be seen as a proof of $(\text{exists } x).(P\ x) \rightarrow B$ since given a term c such that h is a proof of $(P\ c)$, $(g\ c\ h)$ is a proof of B . Given proofs of $(\text{exists } x).(P\ x)$ and $((\text{exists } x).(P\ x) \rightarrow C)$ we should be able to prove C . Thus the proof of $(\text{exists } x).(P\ x)$ should be a function that takes as inputs a proposition C , a proof of $(\text{exists } x).(P\ x) \rightarrow C$ and return a proof of C . Hence we can define $(\text{exists } x).(P\ x)$ as

$(C : *)((x : A)((P\ x) \rightarrow C)) \rightarrow C$. The difference between $((x : A)(P\ x)) \rightarrow C$ and $(x:A)(P\ x) \rightarrow C$ is crucial in this respect. A term of type $((x:A)(P\ x)) \rightarrow C$ is a proof of $((\text{forall } x).(P\ x)) \rightarrow C$ while a term of type $(x:A)(P\ x) \rightarrow C$ is a proof of $(\text{exists } x).(P\ x) \rightarrow C$. Similarly we can define

$A \wedge B$ by $(C : *)((A \rightarrow C) \rightarrow (B \rightarrow C) \rightarrow C) \rightarrow C$ and $A \vee B$ by $(C:*)(A \rightarrow B \rightarrow C) \rightarrow C$. Thus \wedge can be defined as

$[A:][B:](C : *)((A \rightarrow C) \rightarrow (B \rightarrow C) \rightarrow C) \rightarrow C$

and \vee can be defined as

$[A:][B:](C : *) (A \rightarrow B \rightarrow C) \rightarrow C$.

The extension of Fw with 'types dependent on terms' gives us the various versions of Calculus of Constructions. The first version of the Calculus of Constructions, CoC, was presented by T.Coquand in 1985. This logical system allowed powerful axiomatizations, but direct inductive definitions were not possible and induction principles are not provable. The formalism was extended in 1989 by

Coquand and Paulin with primitive inductive definitions, leading to the current Calculus of Inductive Constructions.

1.2 Outline of Thesis :

The organization of the rest of the chapters is as follows.

Chapter 2. Program Development

This chapter is the main chapter. It discusses the development of depth first search and its application for finding strongly connected components.

Chapter 3. Program Extraction

This chapter discusses the realization of axioms and program extraction in Coq. Realization of the two axioms necessary for extraction of depth first search and strongly connected components algorithm is also described.

Chapter 4. Conclusion

APPENDIX

The appendix contains an overview of the current implementation (Coq) of the Calculus of Inductive Constructions.

CHAPTER 2

PROGRAM DEVELOPMENT

This chapter presents the formal development in Coq of the depth first search algorithm and its application for finding strongly connected components. This chapter is divided into three main sections. The section one describes the graph representation used and develops the basic graph manipulation functions. Section two describes the formal development of depth first search algorithm. Section three describes the development of the depth first search algorithm for finding strongly connected components in a directed graph.

2.1 Basic Definitions and Lemmas

2.1.1 Vertices and Vertex Lists

We declare the type V of vertices by

Parameter $V : \text{Set}.$

For checking the equality of vertices, we assume

Axiom $\text{eqV} : (x, y : V) \{ \langle V \rangle x = y \} + \{ \sim \langle V \rangle x = y \}.$

The inductive set of vertex lists is defined by

Inductive Set $\text{list} = \text{nil} : \text{list} \mid \text{cons} : V \rightarrow \text{list} \rightarrow \text{list}$

2.1.2 List Membership

We now define a function to test the membership of a vertex in a list.

Definition occurs_in.

```
Body [l : list][x : V](<Prop>Match l with
  False
  [a : V][m : list][P : Prop]
  (<V>x=a) \ / P).
```

The properties of occurs_in which we prove, for example, include

- a. for any list l and any vertex v, either (occurs_in l v) or $\sim(\text{occurs_in } l \ v)$ holds
- b. a vertex occurs in a list l or a list m iff it occurs in the list obtained by appending m to l
- c. a vertex cannot occur in an empty list.

2.1.3 Removal of a Vertex from a List

For any vertex x, lists l and lx, (rm_V_list x l lx) iff lx is the list obtained by removing all copies of x from l.

```
Inductive Definition rm_V_list[x:V] : list->list->Prop =
  nil_rm_V_list : (rm_V_list x nil nil)
| cons_rm_V_list1 : (y : V)(l : list)(m : list)
  (rm_V_list x l m)->(<V>x=y)->
  (rm_V_list x (cons y l) (cons y m))
| cons_rm_V_list2 : (l : list)(m : list)
  (rm_V_list x l m)->
  (rm_V_list x (cons x l) m).
```

The rm_V_list lemmas which we prove are

- a. for any vertex x and any list l, there exists a list lx such that (rm_V_list x l lx) holds
- b. if (rm_V_list x l lx) then x does not occur in l

- c. if $(rm_V_list \times l \ lx)$ and $(\sim \langle V \rangle x=y)$ and y occurs in l then y occurs in lx .

2.1.4 Graph Definition

In this thesis we consider only directed graphs. The representation we use for graph is

Inductive Definition $graf : Set =$
 $nil_graf : graf$
 $| cons_graf : V \rightarrow list \rightarrow graf \rightarrow graf.$

In this graph representation, we assume that

- a. if a vertex has more than one edge list, the outermost edge list of that vertex in the graph representation is its true edge list.
- b. edge list of a vertex which has no edge list of its own is assumed to be nil.

The reason for assumption (b) is that a vertex can occur in an edge list without having an edge list of its own. For depth first search related algorithms, this assumption is satisfactory. This assumption can be changed to a stronger assumption that if a vertex has no edge list of its own and it occurs in some edge list then its edge list is nil else it has no edge list. Disadvantage of this stronger assumption is an increase in runtime requirement but it has the advantage of distinguishing vertices present in a graph from other vertices.

2.1.5 Edge list of a vertex

In accordance with the assumptions given in the previous section, we define an inductive ternary relation `edges_from_in`. For any graph `G` and vertex `x`, `(edges_from_in x G l)` iff `l` is the edge list of `x` in `G`.

```
Inductive Definition edges_from_in[v1:V] : graf->list->Prop =
  edges_from_nil : (edges_from_in v1 nil_graf nil)
| edges_from_in_1 : (v2 : V)(l1 : graf)(t1 : list)
  (t2 : list) (edges_from_in v1 l1 t1)->(<<V>>v1 = v2) ->
  (edges_from_in v1 (cons_graf v2 t2 l1) t2)
| edges_from_in_2 : (v2 : V)(l1 : graf)(t1 : list)
  (t2 : list) (edges_from_in v1 l1 t1)->(<~<V>>v1 = v2) ->
  (edges_from_in v1 (cons_graf v2 t2 l1) t1).
```

The `edges_from_in` lemmas which we prove are

- a. for any vertex `v` and graph `G`, there exists a list `l` such that `(edges_from_in x G l)` holds.
- b. edge list of a vertex in a graph is unique

2.1.6 Edge Definition

In a graph `G`, there is an edge from vertex `x` to vertex `y` iff `y` occurs in the edge list of `x`. We use this fact in the definition of the relation `edge`.

```
Inductive Definition edge[G:graf;x,y:V] : Prop =
  edge_intro : (l : list)(edges_from_in x G l)->
    (occurs_in l y)->(edge G x y).
```

The lemmas related to edge property which we prove are

- a. there exists no edge between two vertices in a `nil_graf`

- b. the inversion of edge definition, i.e., if there is an edge from x to y in a graph G , then y occurs in the edge list of x .
- c. for any graph G , vertices x, y , and z , and list l , if $(\sim \langle V \rangle x = y)$ then there is an edge from vertex y to vertex z in the graph $(\text{cons_graf } x \ l \ G)$ iff $(\text{edge } G \ y \ z)$.
- d. for any graph G , and vertices x and y , either $(\text{edge } G \ x \ y)$ or $\sim(\text{edge } G \ x \ y)$ holds.

2.1.7 Addition of an Edge to a Graph

If the edge list of a vertex x in a graph G is l , then by assumption (a) in the graph definition, $(\text{cons_graf } x \ (\text{cons } y \ l) \ G)$ is one of the many possible representations of the graph resulting from the addition of an edge from x to y in graph G . We use this representation to define the relation cons_edge_graf . The proposition $(\text{cons_edge_graf } G \ x \ y \ Gxy)$ holds if Gxy is the graph resulting from the addition of an edge from x to y in graph G .

Inductive Definition $\text{cons_edge_graf}[G:\text{graf};x,y:V]:\text{graf} \rightarrow \text{Prop} =$
 $\text{cons_edge_graf_intro} : (l : \text{list})(\text{edges_from_in } x \ G \ l) \rightarrow$
 $(\text{cons_edge_graf } G \ x \ y \ (\text{cons_graf } x \ (\text{cons } y \ l) \ G)).$

The lemmas related to cons_edge_graph which we prove are that for any graphs G and Gxy , and vertices x, y, p , and q

- a. for any graph G and vertices x and y , there exists a graph Gxy such that $(\text{cons_edge_graph } G \ x \ y \ Gxy)$ holds.
- b. if $(\text{cons_edge_graf } G \ x \ y \ Gxy)$ then $(\text{edge } Gxy \ x \ y)$.

- c. if $(\text{cons_edge_graf } G \ x \ y \ Gxy)$ and $(\text{edge } G \ p \ q)$ then $(\text{edge } Gxy \ p \ q)$.

2.1.8 Composition of Two Graphs

By composition of two graphs G_a and G_b , we mean the graph G_{ab} obtained by the additions of edge in G_b to G_a . We characterize the composition of two graphs by the ternary relation union_graf . The definition of union_graf uses the assumption (a) of the graph definition.

```
Inductive Definition union_graf[G:graf] : graf->graf->Prop =
  nil_union_graf : (union_graf G nil_graf G)
| cons_union_graf : (x : V)(la : list)(lb : list)
  (Gb : graf)(Gc : graf)
  (union_graf G Gb Gc)->(edges_from_in x G la)->
  (union_graf G (cons_graf x lb Gb)
    (cons_graf x (app la lb) Gc)).
```

The lemmas related to union_graf which we prove are

- a For any graphs G_a and G_b , there exists a graph G_{ab} such that $(\text{union_graf } G_a \ G_b \ G_{ab})$ holds.
- b For any graphs G_a , G_b , and G_{ab} , and vertices x and y , if $(\text{union_graf } G_a \ G_b \ G_{ab})$ then $(\text{edge } G_{ab} \ x \ y)$ iff either $(\text{edge } G_a \ x \ y)$ or $(\text{edge } G_b \ x \ y)$.

2.1.9 Removal of a Vertex from a Graph

For any graphs G , and G_x , and vertex x , the relation $(\text{rm_V_graf } x \ G \ G_x)$ holds if G_x is the graph obtained by removing the edge list of x and also v from all other edge lists of the graph G .

```

Inductive Definition rm_V_graf[x:V] : graf->graf->Prop =
  nil_rm_V_graf      : (rm_V_graf x nil_graf nil_graf)
| cons_rm_V_graf1    : (Ga : graf)(Gb : graf)
  (l : list)(rm_V_graf x Ga Gb)->
  (rm_V_graf x (cons_graf x l Ga) Gb)
| cons_rm_V_graf2    : (y : V)(Ga : graf)(Gb : graf)
  (la : list)(lb : list)(rm_V_list x la lb)->
  (~(<V>x=y))->(rm_V_graf x Ga Gb)->
  (rm_V_graf x (cons_graf y la Ga) (cons_graf y lb Gb)).

```

The lemmas of the property `rm_V_graf` which we prove are

- a. for any vertex `x` and graph `G` there exists graph `Gx` such that `(rm_V_graf x G Gx)` holds.
- b. if `(rm_V_graf x G Gx)` then there is no edge from `x` or to `x` in graph `Gx`.

2.1.10 The Set `pair_dfs` and its Projection Functions

We end this section with the definition of `pair_dfs`. A term of type `pair_dfs` has three components, two graphs and a list. The set `pair_dfs` should not be confused with a pair type since it has actually three components. We also define the three destructors `pd_grafa`, `pd_grafb` and `pd_list` to access the three components.

```

Inductive Definition pair_dfs : Set =
  pair_dfs_intro : (ga,gb : graf)(l : list)pair_dfs.

```

Definition `pd_grafa`.

```

Body [d : pair_dfs]<<graf>Match d with
  [ga : graf][gb : graf][l : list]ga>.

```

Definition `pd_grafb`.

```

Body [d : pair_dfs]<<graf>Match d with
  [ga : graf][gb : graf][l : list]gb>.

```

Definition `pd_list`.

```

Body [d : pair_dfs]<<list>Match d with
  [ga : graf][gb : graf][l : list]l>.

```

2.2 Development of Depth First Search

2.2.1 Depth First Search Algorithm

A depth first search (dfs_fn) call takes as input a graph(G), a start vertex (start), a list of vertices (v_list), and a list (list_out) of vertices reached so far. It returns as output a term p_out of type pair_dfs whose components are a graph (G_out) of vertices not yet reached, a graph (tree_out) which is the depth first search tree of vertices reached in the current call, and a new list (list_out) of vertices reached so far.

```

define dfs_fn (G,start,v_list,list_out)
{
  let G_out          = the graph resulting from the removal of
                        start from G
  let tree_out       = nil_graf
  let list_out       = (cons start list_out)
  (* the above initialization steps form the basis of*)
  (* init_dfs in the definition of dfs relation *)
  for each v in v_list do
  {
    if v occurs_in list_out
    then {} (*then part forms the basis for cons_dfsa*)
    else
    {
      let l      = edge list of v in G_out
      let p      = dfs_fn (G_out v l list_out)
      G_out      = (pd_grafa p)
      tree_out   = graph resulting from the addition of
                    an edge from start to v to the union
                    of graphs tree_out and (pd_grafb p)
      list_out   = (pd_list p)
    }
    (*else part forms the basis for cons_dfsb*)
  }
}
return (pair_dfs_intro G_out tree_out list_out)
}

```

In the above algorithm G_out and list_out are better seen as 'tail recursive' variables - which is not the case for tree_out. Now the depth first search tree rooted at start in graph G is (pd_grafa

(dfs_fn (G start v_list nil))) if edge list of vertex v in graph G is v_list. To find the depth first search forest for a graph G, call (pd_grafa (dfs_fn (G start v_list nil))) where start is a vertex not in graph G, and v_list is the list of all vertices in graph G.

2.2.2 The dfs Relation

We now give the inductive definition (dfs) of the relationship that should hold between the arguments and the result term of the dfs_fn given above.

```
Inductive Definition dfs: graf->V->list->list->
      pair_dfs->Set =
  init_dfs : (gin,gout : graf)(vert : V)(Qlin : list)
    (rm_V_graf vert gin gout)->
    (dfs gin vert nil Qlin
      (pair_dfs_intro gout nil_graf (cons vert Qlin)))
  | cons_dfsa : (gin : graf)(verta,vertb : V)(Qlin : list)
    (l: list)(p : pair_dfs)(dfs gin verta l Qlin p)->
    (occurs_in (pd_list p) vertb)->
    (dfs gin verta (cons vertb l) Qlin p)
  | cons_dfsb : (gin,Qgab,Qgc : graf)(verta,vertb : V)
    (Qlin : list)(la,lb : list)(pa,pb : pair_dfs)
    (dfs gin verta la Qlin pa)->
    (~ (occurs_in (pd_list pa) vertb))->
    (edges_from_in vertb (pd_grafa pa) lb)->
    (dfs (pd_grafa pa) vertb.lb (pd_list pa) pb)->
    (union_graf (pd_grafb pa) (pd_grafb pb) Qgab)->
    (cons_edge_graf Qgab verta vertb Qgc)->
    (dfs gin verta (cons vertb la) Qlin
      (pair_dfs_intro (pd_grafa pb) Qgc (pd_list pb))).
```

2.2.3 Well Founded Induction for Graphs

We need a well_founded induction rule for graphs. For this purpose we define a subgraph relationship for graphs. If from a graph G, a vertex v with non nil edge list is removed to get a graph Gv, then Gv is a proper subgraph of G. Deleting any vertex from any proper

subgraph of G results in further subgraphs of G . By this definition, a graph in which all edge lists are nil does not have any proper subgraph. We also prove the transitive property of subgraph relation.

```

Inductive Definition subgraf[G : graf] : graf->Prop =
  subgraf_introa : (Gs : graf)(x : V)(l : list)
    (edges_from_in x G l)->(~<list>nil=l)->
    (rm_V_graf x G Gs)->(subgraf G Gs)
| subgraf_introb : (Gs,Gss : graf)(x : V)
    (subgraf G Gs)->(rm_V_graf x Gs Gss)->
    (subgraf G Gss).

```

Our well founded induction rule now states that if a statement is true for all subgraphs of a graph implies that it is true for that graph also, then that statement holds for all graphs. If we are to use this induction principle to prove a statement for all graphs, then that statement, in the base case, must hold for all graphs with nil edge lists. Graphs with nil edge lists are nil_graf and all graphs (cons v nil G) such that v is any vertex and G a graph with nil edge lists.

We use the variable well_founded and the axiom of well_founded_recursion given in merge_list.v in Coq documentation. To use the axiom of well_founded recursion for graphs we assume the axiom graf_wf.

```

Variable well_founded : (A:Set)(A->A->Prop)->Prop.

```

```

Axiom well_founded_recursion : (A:Set)(R:A->A->Prop)
  (well_founded A R)->(P:A->Set)
  (a:A)((b:A)((c:A)(R c b)->(P c))->(P b))->(P a).

```

```

Axiom graf_wf:(A : Set)(f:A->graf)
  (well_founded A [a1,a2:A](subgraf (f a2)(f a1))).

```

For any Set type A, $(\text{well_founded } A \ R)$ holds if R is a relation which defines the well_founded ordering for terms of type A. Our assumption graf_wf states that for any Set type A, and function f from A to graph, $(\text{well_founded } A \ \text{graf_pred})$ holds, where $(\text{graf_pred } x \ y)$ means $(f \ y)$ is a subgraph of $(f \ x)$. Axiom $\text{well_founded_recursion}$ defines a well_founded induction rule for every $(\text{well_founded } A \ R)$. Our axiom of graf_wf is very general. For the proofs that follow, we need only $(\text{well_founded } \text{graf} \ [x,y:\text{graf}](\text{subgraf } y \ x))$.

Hence to specialize graf_wf , we define an identity function for graphs.

Definition id_graf .
 Body $[a : \text{graf}] a : \text{graf} \rightarrow \text{graf}$.

2.2.4 Statement and Proof of Main Goal

Now our main goal is to prove that for any graph G and vertex v , if edges from v in G is l , then there exists p of type pair_dfs such that $(\text{dfs } G \times l \ \text{nil } p)$ holds. For this purpose we define a relation dfs_spec and state the goal based on dfs_spec . The motivation for dfs_spec is that it helps us to hide irrelevant details, like existence of l such that $(\text{edges_from_in } x \ G \ l)$, from the goal. Purpose of this hiding is that program extracted from the proof of our goal will find such an l instead of l also being an input to the program. (We have to prove the existence of such an l with in the proof of our goal.)

Inductive Definition $\text{dfs_spec}[G:\text{graf};x:V;p:\text{pair_dfs}]:\text{Prop} =$
 $\text{dfs_spec_intro}:: (l : \text{list})(\text{edges_from_in } x \ G \ l) \rightarrow$
 $(\text{dfs } G \times l \ \text{nil } p) \rightarrow (\text{dfs_spec } G \times p).$

```
Goal (G : graf)(x : V)
  {p : pair_dfs l (dfs_spec G x p)}.
```

To prove the above goal, we prove the following lemma.

```
Goal (gin : graf)(x : V)(l : list)(Qlin : list)
  (edges_from_in x gin l)->(~<list>nil=l)->
  (exist_sp pair_dfs
    [p:pair_dfs](dfs gin x l Qlin p)
    [p:pair_dfs](subgraf gin (pd_grafa p)))).
```

This lemma states that for any graph *gin* and vertex *x*, if edges from *x* in *gin* is *l* such that *l* is not nil then for any list *Qlin* there exists *p* of type *pair_dfs* such that *(dfs gin x l Qlin p)* and *(subgraf gin (pd_grafa p))* holds. *Qlin* is informally the list of vertices already reached. The motivation for the lemma is that we have to prove by well founded induction and only removal of a vertex with non nil edge list gives a subgraph. The induction outline is informally as follows. We remove the start vertex *v* to get the graph of vertices not yet reached (*g_out*). The graph *g_out* is proved to be a proper subgraph of *gin*. Then for each vertex *vl* in *l* we prove that either case (a) or case (b) given below holds.

- a. if edges from *vl* in *g_out* is nil then we remove *vl* to get the new *g_out*. We use the property that the old *g_out* is a proper subgraph of *gin* to prove that new *g_out* is a proper subgraph of *gin*. (Removal of any vertex from a proper subgraph gives another subgraph.)
- b. if edges from *vl* in *g_out* is not nil, then we prove that *g_out* returned by the recursive call is a proper subgraph of *gin*. The new *g_out* is a proper subgraph of old *g_out* by induction

hypothesis and old g_out we have already proved is a proper subgraph of g_{in} . Thus transitive property of subgraphs give us the required proof.

The lemma goal statement above had used a relation $exist_sp$ to state the existence of term p of type $pair_dfs$ such that $\langle P\ p \rangle$ and $\langle Q\ p \rangle$ holds, where P is of type $pair_dfs \rightarrow Prop$ and Q of type $pair_dfs \rightarrow Set$. We now give the definition of $exist_sp$.

Inductive Definition $exist_sp[A:Set;P:A \rightarrow Set;Q:A \rightarrow Prop]:Set =$
 $exist_sp_intro : \langle x : A \rangle \langle P\ x \rangle \rightarrow \langle Q\ x \rangle \rightarrow \langle exist_sp\ A\ P\ Q \rangle.$

The proof of our main goal now follows easily. We prove separately the case where edge list of x in G is nil . For the case when edge list is not nil we use the lemma proved above.

2.3 Depth First Search for Strongly Connected Components

2.3.1 Definitions and Lemmas

We define a predicate min . For any $n1$, $n2$, and $n3$ of type nat , $\langle min\ n1\ n2\ n3 \rangle$ stands for the statement that $n3$ is the minimum of $n1$ and $n2$. We prove that for any $n1$ and $n2$ there exists $n3$ of type nat such that $\langle min\ n1\ n2\ n3 \rangle$ holds.

Inductive Definition $min[x,y : nat] : nat \rightarrow Prop =$
 $min_le : \langle le\ x\ y \rangle \rightarrow \langle min\ x\ y\ x \rangle$
 $| min_gt : \langle gt\ x\ y \rangle \rightarrow \langle min\ x\ y\ y \rangle.$

We also prove that, for any $n1$ and $n2$ of type nat either $n1$ is equal to $n2$ or $n1$ is not equal to $n2$.

We define a new Set type ll , the type of list of list of vertices.

This type is necessary to represent a collection of strongly connected components.

```
Inductive Definition ll : Set =
  nil_ll : ll
| cons_ll : list -> ll -> ll.
```

We now define the relation `rm_l_l`. For any lists `la`, `lb`, and `lc` `(rm_l_l la lb lc)` holds if `lc` is the list obtained by removing vertices in `la` from `lb`. For a given `la` and `lb`, we also prove the existence of such an `lc`.

```
Inductive Definition rm_l_l[l:list]:list->list->Prop =
  rm_l_l_nil : (rm_l_l l nil nil)
| rm_l_l_consa : (la,lb : list)(v : V)(rm_l_l l la lb)->
  (occurs_in l v)->(rm_l_l l (cons v la) lb)
| rm_l_l_consb : (la,lb : list)(v : V)(rm_l_l l la lb)->
  (~ (occurs_in l v))->(rm_l_l l (cons v la) (cons v lb)).
```

2.3.2 Depth First Number and Associated Definitions

Depth first search for strongly connected components (`dfs_scc`) requires depth first numbering of vertices. The `dfs` algorithm always keep track of vertices reached so far in a list, in the order in which vertices where reached. Hence if we see this list as a stack, then the height of a vertex in the stack gives its depth first search number. We now define a relation `depth` such that `(dfs x l nat1 nat2)` holds if `nat1` is the size of list `l` and `nat2` the height of vertex `x` when `l` is seen as a stack. We also define a relation `depth_spec`, such that `(depth_spec x l nat2)` holds if there exists `nat1` such that `(depth x l nat1 nat2)` holds. The definition of `depth_spec` helps us to hide `nat1` from our goal, which is to prove that there exists `nat2` such that it is the depth of `x` in `l`.

```

Inductive Definition depth[x:V]:list->nat->nat->Prop =
  depth_nil    : (depth x nil 0 0)
| depth_consa  : (l : list)(m,n : nat)
  (depth x l m n)->
  (depth x (cons x l) (S m) (S m))
| depth_consb  : (l : list)(v : V)(m,n : nat)
  (depth x l m n)->(~<V>x=v)->
  (depth x (cons v l) (S m) n).

```

```

Inductive Definition depth_spec[x:V;l:list]:nat->Set=
  depth_spec_intro : (m,n : nat)(depth x l m n)->
    (depth_spec x l n).

```

The depth_min_fn algorithm is informally given below. The dfs_list argument in the algorithm should be seen as a stack of vertices reached so far in the process of depth first search. The basic aim of the algorithm is to return the minimum value among default value and the depth of vertices of l (that occurs in check_list) in dfs_list.

```

define depth_min_fn (check_list : list; default : nat;
                    dfs_list : list; l : list)
{
  let return_num = default
  for each v in l do
  {
    if v occurs in check_list
      then return_num = minimum of return_num and
                        depth of v in dfs_list
    else {}
  }
  return (return_num)
}.

```

We now give the inductive definition (depth_min) of the relationship that should hold between the arguments and the result term of depth_min_fn. We also prove the existence of the depth_min_fn.

```

Inductive Definition depth_min
  [lis:list;def:nat;dfs_list:list] :list->nat->Prop=
  depth_min_nil : (depth_min lis def dfs_list nil def)
| depth_min_consa : (l : list)(v : V)(m,n,o:nat)
  (depth_min lis def dfs_list l m)->(occurs_in lis v)->

```

```

(depth_spec v dfs_list n)->(min m n o)->
(depth_min lis def dfs_list (cons v l) o)
| depth_min_consb : (l : list)(v : V)(m:nat)
(depth_min lis def dfs_list l m)->
(~(occurs_in lis v))->
(depth_min lis def dfs_list (cons v l) m).

```

2.3.3 The Set pair_scc and its Projection Functions

We define a Set type pair_scc, a term of type pair_scc has three components a nat, a list of vertices and a list of list of vertices. To project out these components from a pair_scc term, we also define three projection functions.

```

Inductive Definition pair_scc : Set =
pair_scc_intro : nat->list->ll->pair_scc.

```

```

Definition ps_num.
Body [p : pair_scc](⟨nat⟩Match p with
[a : nat][c : list][d : ll]a).

```

```

Definition ps_list.
Body [p : pair_scc](⟨list⟩Match p with
[a : nat][c : list][d : ll]c).

```

```

Definition ps_ll.
Body [p : pair_scc](⟨ll⟩Match p with
[a : nat][c : list][d : ll]d).

```

2.3.4 Strongly Connected Components Algorithm

In a depth first search call for strongly connected components, the first argument is a graph (original_graf) which is always passed unchanged to all recursive calls. The next four arguments are same as that for depth first search (dfs_fn) call. They are a graph (G), a start vertex (start), a list (la) of vertices and a list (list_out) of vertices reached so far. The fifth new argument is psa of type pair_scc. The first component of psa, (ps_num psa), is the number of vertices reached so far. The second component of psa,

(ps_list psa) is a stack of vertices. The main use of this stack is to store vertices, not yet a complete strongly connected component(scc), in the order in which they were reached. The third component of psa, (ps_ll), is the list of strongly connected components collected so far.

```

define dfs_scc_fn (original_graf,G,start,la,list_out,psa)
  (* returns three values : a term of type pair_dfs
    a term of type pair_scc and
    a term of type nat*)
(
let G_out = the graph resulting from the removal of start from G
let tree_out      = nil_graf
let list_out      = (cons start list_out)
let num_out       = (S (ps_num psa))
let stack_out     = (cons start (ps_list psa))
let ll_out        = (ps_ll psa)
let minnata       = (ps_num psa)
  (*the above initialization steps form the basis of*)
  (*init_dfs_scc in the definition of dfs_scc relation*)
for each v in la do
  (
    if v occurs_in list_out
      then
        (
          (*then part forms the basis for cons_dfs_scc1*)
          minnata = depth_min_fn (stack_out minnata
                                list_out (cons v nil))
        )
      else
        (
          let lb      = edge list of v in G_out
          let (p,psc,minnatb)
            = dfs_scc_fn (original_graf G_out v lb list_out
                        (pair_dfs_intro num_out stack_out
                          ll_out))

          G_out      = (pd_grafa p)
          tree_out   = graph resulting from the addition of an
                        edge from start to v to the union of
                        graphs tree_out and (pd_grafb p)
          list_out   = (pd_list p)
          let lc      = edge_list of v in original_graf
          minnatb = depth_min_fn (stack_out minnatb list_out lc)
          if (minnatb == num_out)
            then
              (
                (*then part forms the basis for cons_dfs_scc2*)
                num_out = (ps_num psc)
                let lis = list obtained by removing vertices in

```

```

        stack_out from (ps_list psc)
        (*can be seen as popping (ps_list psc) until
          it becomes equal to stack_out*)
        ll_out = (cons_ll lis (ps_ll psc))
      }
    else
      {
        (*else part forms the basis for cons_dfs_scc3*)
        num_out = (ps_num psc)
        stack_out = (ps_list psc)
        ll_out = (ps_ll psc)
        minnata = minimum of minnata and minnatb
      }
  }
return ((pair_dfs_intro G_out tree_out list_out)
        (pair_scc_intro num_out stack_out ll_out)
        minnata)
}.

```

2.3.5 The dfs_scc Relation

We now give the definition of the relationship (dfs_scc) that should hold between the result terms and arguments of the dfs_scc_fn given above. Informally (dfs_scc original_graf G start la list_out p psa psb minnat) holds if (p psb minnat) = dfs_scc_fn (original_graf G start la list_out psa). The relation is based on the introduction rules for depth first search relation. For each dfs introduction rule we specify the extra logical relations that should hold to prove (dfs_scc original_graf G start la list_out p psa psb minnat).

```

Inductive Definition dfs_scc[Gin:graf]
: graf->V->list->list->pair_dfs->
  pair_scc->pair_scc->nat->Prop =
init_dfs_scc : (gin,gout,Qg : graf)(vert : V)(Qlin : list)
  (ps : pair_scc) (rm_V_graf vert gin gout)->
  (dfs_scc Gin gin vert nil Qlin
   (pair_dfs_intro gout Qg (cons vert Qlin)) ps
   (pair_scc_intro (S(ps_num ps)) (cons vert (ps_list ps))
    (ps_ll ps)) (ps_num ps))
cons_dfs_scc1 : (gin : graf)(verta,vertb : V)
  (Qlin:list)(l : list)(minnat,newmin : nat)
  (p:pair_dfs)(pin,pout : pair_scc)

```

```

(dfs_scc Gin gin verta l Qlin p pin pout minnat)->
(occurs_in (pd_list p) vertb)->
(depth_min (ps_list pout) minnat (pd_list p)
  (cons vertb nil) newmin)->
(dfs_scc Gin gin verta (cons vertb l)
  Qlin p pin pout newmin)
| cons_dfs_scc2 : (gin,Qg:graf)(verta,vertb:V)
  (Qlin,la,lb,lc:list)(pa,pb : pair_dfs)
  (psa,psb,psc : pair_scc)(minnata,minnatb,newmin : nat)
  (dfs_scc Gin gin verta la Qlin pa psa psb minnata)->
  (~ (occurs_in (pd_list pa) vertb))->
  (edges_from_in vertb (pd_grafa pa) lb)->
  (dfs_scc Gin (pd_grafa pa) vertb lb (pd_list pa)
    pb psb psc minnatb)->
  (edges_from_in vertb Gin lc)->
  (depth_min (ps_list psb) minnatb
    (pd_list pb) lc newmin)->
  (<nat>(ps_num psb)=newmin)->
  (rm_l_1 (ps_list psb) (ps_list psc) lis)->
  (dfs_scc Gin gin verta (cons vertb la) Qlin
    (pair_dfs_intro (pd_grafa pb) Qg (pd_list pb))
    psa (pair_scc_intro (ps_num psc)
      (ps_list psb) (cons ll lis (ps_ll psc)))) minnata)
| cons_dfs_scc3 : (gin,Qg:graf)(verta,vertb:V)
  (Qlin,la,lb,lc:list)(pa,pb : pair_dfs)
  (psa,psb,psc : pair_scc)(minnata,minnatb,newmin,nm : nat)
  (dfs_scc Gin gin verta la Qlin pa psa psb minnata)->
  (~ (occurs_in (pd_list pa) vertb))->
  (edges_from_in vertb (pd_grafa pa) lb)->
  (dfs_scc Gin (pd_grafa pa) vertb lb (pd_list pa)
    pb psb psc minnatb)->
  (edges_from_in vertb Gin lc)->
  (depth_min (ps_list psb) minnatb
    (pd_list pb) lc newmin)->
  (~ (<nat>(ps_num psb)=newmin))->
  (min minnata newmin nm)->
  (dfs_scc Gin gin verta (cons vertb la) Qlin
    (pair_dfs_intro (pd_grafa pb) Qg (pd_list pb))
    psa psc nm).

```

2.3.6 Statement and Proof of Main Goal

We now prove the following lemma.

Goal (Gin,gin : graf)(x : V)(la : list)(Qlin : list)
 (p:pair_dfs)(dfs gin x la Qlin p)->(psa:pair_scc)
 {psb:pair_scc&(nm:nat)(dfs_scc Gin gin x la Qlin
 p psa psb nm)}).

The lemma states that for all p of type pair_dfs whenever (dfs gin

$x \text{ la } Q \text{ lin } p$) holds then for all psa of type `pair_scc` there exists psb of type `pair_scc` and nm of type `nat` such that $(dfs_scc \ G \ \text{gin} \ \text{gin} \ x \ \text{la} \ Q \ \text{lin} \ p \ \text{psa} \ \text{psb} \ nm)$ holds. The main proof step is the elimination of $(dfs \ \text{gin} \ x \ \text{la} \ Q \ \text{lin} \ p)$ which reduces the problem to three subgoals corresponding to the three depth first search introduction rules.

For defining the relationship between the `graf` G , vertex x and the collection of strongly connected components obtained by a depth first search rooted at x we have to hide the irrelevant information in $(dfs_scc \ G \ G \ x \ \text{la} \ \text{nil} \ p \ (\text{pair_scc_intro} \ (S \ 0) \ \text{nil} \ \text{nil_ll}) \ \text{psb} \ nm)$ such as nm and the fact that edges from x in G is la . Also the vertices remaining in the stack $(ps_list \ \text{psb})$ form the final strongly connected component. For this purpose we define the `dfs_scc_spec` relationship.

```
Inductive Definition dfs_scc_spec[g:graf;x:V;lla:ll]:Prop =
  dfs_scc_spec_intro : (p : pair_dfs)(ps : pair_scc)
    (mn : nat)(la : list)(edges_from_in x g la)->
    (dfs_scc g g x la nil p
      (pair_scc_intro (S 0) nil nil_ll) ps mn)->
    (<ll>lla = (cons_ll (ps_list ps) (ps_ll ps)))->
    (dfs_scc_spec g x lla).
```

Now our main goal will be to prove

```
(G : graf)(x : V){lla : ll | (dfs_scc_spec G x lla)}.
```

To prove this we first prove the existence of an la such that edges from x in G is la . The main steps of the proof then boils down to an induction on la and elimination on the lemma given above.

CHAPTER 3

PROGRAM EXTRACTION

The constructive proof of a specification has two parts, a logical part and a computational part. The logical part is the correctness proof necessary for proving that the computational part actually meets the specification. An example of this logical part is the proof for $\sim(x \leq y) \rightarrow (x > y)$ that comes often in the constructive proof for an arithmetical function.

The program extractor extracts the computational content (program) from the constructive proof of a specification. It is based on a realizability interpretation that relates the extracted program to the original specification. The interest of a realizability interpretation is not just to make sure that the extracted program is correct but also to give the possibility of interpreting axioms. This is as follows.

Assume that there is program p correct with respect to a specification P and that we develop a proof q of a specification Q under the assumption P . The term t extracted from q is not closed and cannot be executed. Now if we substitute the occurrences of the assumption in the term t by the program p , we get a closed term correct with respect to the specification Q . This process of substitution is referred to as realization of axioms.

Thus we may use assumptions which are not provable but for which a correct program can be found. We may also use logical assumptions in our proof which we need not realize.

In the extraction of the depth first search algorithm and the algorithm for finding strongly connected components we have to realize 2 axioms. The set V of vertices is instantiated by the FML int type with the command `Instantiate V Int`. Now the axiom `eqV` which states that equality for vertices is decidable is instantiated by `Realize eqV [x,y] if x=y then left else right`.

Finally the axiom of well founded recursion is realized by

```
Realize well_founded_recursion
  [x,f](let rec F = [y](f y F) in (F x)).
```

The resulting programs have a satisfactory computational behavior.

CHAPTER 4

CONCLUSION

We have defined logically the relationship that should hold between the input arguments and the output term for depth first search algorithm and also for depth first search algorithm for finding strongly connected components. This reduces the logical specification of these algorithms to statements of the form for any input term there exist an output term such that the (respective) relationship holds. We extracted the algorithms that meet these specifications from their constructive proofs in the Coq programming environment.

The relationship used for the specification of strongly connected components is based on depth first search. What remains to be done is to prove a high level property of this relationship, which can proceed as follows. Define the relationship `scc` of two arguments, a graph and a list, which holds true if the vertices of the list forms a strongly connected component of the graph. Then the high level property of the strongly connected components program development reduces to the statement that the vertices of a graph (G) can be split into lists such that each list l satisfies $(scc\ G\ l)$.

The difficulty faced in the attempt for a proof of the above property is that proof techniques do not scale up properly. In the case of functions where a high level specification can be proved by simple induction, the resulting proofs are simple and elegant. However, even for trivial algorithms the proof size as well as the

subgoal size can become too overwhelming. Choosing the right induction hypothesis as well as proving is largely a trial and error operation. For example, in some cases it was found easier to prove a general statement and then specialize it rather than directly prove the specialized statement. Some techniques of proving, e.g., writing functions that return propositions for proving inversion of definitions and induction without introducing all variables into the context are often useful. A collection of heuristic rules to help in the proving process can help in reducing the time spent in trial and error.

APPENDIX 1

An Overview of the Coq System

In the Coq system, every correct term has a type, the type itself has a type called a sort. There are two sorts of types : Prop and Set. An object whose type is a Prop is called a proposition and an object whose type is Set is called a specification.

The main constructions may be briefly described as follows[7].

- a. names refer to declared variables or declared constants
- b. $(M\ N)$ denotes the application of functional object M to object N.
- c. $[x:T].M$ abstracts the variable x of type T to form a functional object.
- c. $\text{Ind}\{X:S|C_1,\dots,C_n\}$ defines inductively a set or a proposition, according to sort S with n constructors of type C_1,\dots,C_n .

Further cases refer to the constructors of an inductive type, and its elimination principles corresponding to an induction principle for propositions, and a recursion principle for sets. A high level language, called the mathematical vernacular, permits to name these conventions conveniently.

This Appendix is divided into two main sections.

Section 1 contains a description of the Tactics Theorem prover.

Section 2 contains a description of inductive definitions.

Section 3 contains an of a proof with Tactics theorem prover.

1. The Tactics Theorem Prover

The Tactics Theorem Prover is a goal directed inference engine, in the spirit of Prolog, but with a proof search mechanism driven by the method of tactics pioneered by the LCF system[11]. The principle used is "Truth is Inhabitation", so that to prove proposition is to build an element of type this proposition.

A tactic is a function that takes a goal $Q \vdash C$ where Q is a context and C the desired conclusion and returns a list of subgoals $(Q_1 \vdash C_1, \dots, Q_n \vdash C_n)$ and a validation function f such that if y_1, \dots, y_n are respectively the proofs of $Q_1 \vdash C_1, \dots, Q_n \vdash C_n$ then $(f \ y_1 \ \dots \ y_n)$ is a proof of $Q \vdash C$.

We shall now briefly illustrate the five basic families of tactics.

The first class of tactics concern parameter introductions. The tactic 'Intro H.' when applied to a goal of the form $Q \vdash A \rightarrow B$, returns the subgoal

$Q; H:A \vdash B$ and implicitly a validation function f which takes a proof y of the subgoal and returns $([H].y)$. The tactic 'Intro H.' when applied to a goal of the form

$Q \vdash (x:A) \rightarrow (P \ x)$ returns the subgoal $Q; H:A \vdash (P \ H)$ and implicitly a validation function f which takes a proof y of the subgoal and

returns $([H:A]y)$. In the above two cases the variable name H should be such that it is not already the name of some assertion in the context Q .

The second class of tactics is that of resolution tactics. If in the context Q , H is the proof of statement P , then the tactic 'Apply H ' tries to resolve the current goal with the conclusion of P . If the matching is successful then Apply generates as many subgoals as there are premises in P . The tactic 'Apply H ' when applied to a goal of the form $Q; H: A \rightarrow B \rightarrow C \mid - C$, returns the subgoals $Q; H: A \rightarrow B \rightarrow C \mid - A$ and $Q; H: A \rightarrow B \rightarrow C \mid - B$ such that if their proofs are y_1 and y_2 respectively then $(H \ y_1 \ y_2)$ is a proof of the main goal.

The third class is that of elimination tactics. A brief description of elimination tactics is given in the section on inductive definitions.

The fourth class is that of convertibility tactics. The basic convertibility tactics are Change and Unfold. An example of Fold tactic is given in section 3. The tactic 'Change H .' when applied to a goal G , replaces goal G by H , if H is convertible to G by Beta rule or elimination rules for inductive terms.

The fifth class is that of automatic search tactics. The tactic Auto tries successively to apply, by order of priority, the axioms in Hint list, until finding a complete proof, with a maximal search depth of 5. Theorems can be added into the Hint list with the command Hint.

2. Inductive Definitions

Inductive definitions allow an internal representation of data types, inductive predicates and logical connectives[12]. An

inductively defined type is given by a complete list of constructors for that type. We reason about the type with an appropriate induction principle, we write programs over the type using iteration and develop programs using the recursion principle. Induction principles as well as recursion principles are not provable in CoC. In Coq, these principles are automatically generated from the definitions and introduced into the context. The system also realizes automatically the recursion principle with primitive recursive operators. An illustration of the various applications of inductive definitions are illustrated below with natural numbers.

The inductive Set of unary natural numbers can be defined by

```
Inductive Set nat = 0 : nat | S : nat->nat.
```

This defines together with the inductive Set, bound to name nat, its two constructors, bound to names 0 and S, an induction principle (nat_ind) of type the proposition:

```
(P:nat->Prop)(P 0)->((x:nat)(P x)->(P (S x)))->(n:nat)(P n).
```

and a recursion rule (nat_rec), of type the specification:

```
(P:nat->Set)(P 0)->((x:nat)(P x)->(P (S x)))->(n:nat)(P n).
```

The type of nat_ind states if (P 0) is true and if (P (S x)) is true whenever (P x) is true then P : nat->Prop is true for all n. Similarly nat_rec states that if (P 0) can be computed and if (P (S x)) can be computed whenever (P x) can be computed the (P x) can be computed for all x of type nat.

As an example inductive definition of functions, the definition of a function for the addition of two natural numbers is given below.

```

Definition plus : nat->nat->nat=
  [n,m:nat](<nat>Match n with
    (*0*) m
    (*S p*) [p:nat][result:nat](S result))

```

It can be seen from this example that Match corresponds to higher type primitive recursion. An example for the definition of an inductive predicate is given below.

```

Inductive Definition le : nat->nat->Prop =
  le_0 : (n : nat)(le 0 n)
  | le_S : (n,m : nat)->(le n m)->(le (S n) (S m)).

```

The inductive definition of le introduces into the context two constructors le_0 and le_S. The constructor le_0 takes as input a number n and returns a term of type (le 0 n). The constructor le_S takes as input numbers n,m and a term of type (le n m) and returns a term of type (le (S n) (S m)). It also introduces into the context an induction principle le_ind, based on the fact that le is the smallest relation closed with respect to the two constructors. The type of le_ind is

```

(R:nat->nat->Prop)
((n:nat)(R 0 n))->
((n,m:nat)(le n m)->(R n m)->(R (S n) (S m)))->
(n,m:nat)(le n m)->(R n m)

```

An example application of the induction principle is given below.

Suppose we have to prove

(n,m:nat)(le n m)->(Q n m) for Q:nat->nat->Prop. Then we introduce the parameters into the context to get a goal of the form

```

      (Q n m)
=====
H : (le n m)
m : nat
n : nat

```

Now we use the induction principle for `le`. Instead of directly using `le_ind` in the proof we use the tactic

`Elim H`. The application of this tactic result in a second order matching of the conclusion $(R\ n\ m)$ of the induction principle with the goal $(Q\ n\ m)$. If this matching is successful then it will result in subgoals corresponding to the premises of `le_ind`. In the present example, the subgoals will be

$(n:\text{nat})(Q\ 0\ n)$ and

$((n:\text{nat})(m:\text{nat})(le\ n\ m) \rightarrow (Q\ n\ m) \rightarrow (Q\ (S\ n)\ (S\ m)))$.

If we have proved the basic properties of the `le` predicate, then we can put them into effective use in proving these two subgoals. Thus to prove a goal G in context Q , it may be easier to prove that $(le\ n\ m)$ holds in Q , and then eliminate $(le\ n\ m)$ to get less complex subgoals. This example clearly illustrates the importance of inductive definitions. Inductive definitions in Coq help us to introduce and realise automatically consistent recursion (induction) principles. This often leads to elegant proofs, since we can split a goal into less complex goals easily.

3. A Proof with Tactics Theorem Prover

This section contains a version in Coq of the constructive proof given by Backhouse[2] for the statement that principle of excluded middle, though not provable, is not false in constructive logic.

Goal (A : Prop)~~(A \ / ~A).

(We use the tactic Unfold replace ~ by its definition. The * in 'Unfold * not' indicated that all occurrences of * in goal should be replace by its definition.)

```
Coq < Unfold * not.
1 subgoal
  (A:Prop)((A\ / A->False)->False)->False
```

(We now introduce all the parameters into the context.)

```
Coq < Intros.
1 subgoal
  False
  =====
  H : (A\ / A->False)->False
  A : Prop
```

(We apply the assumption H to prove the goal. Since H states that if (A\ / A->False) is true then False is true, Our new goal will be to prove A\ / (A->False) is true.)

```
Coq < Apply H.
1 subgoal
  A\ / A->False
  =====
  H : (A\ / A->False)->False
  A : Prop
```

(Now we have to prove either A is true or A->False is true. We use the tactic Right to indicate that we want to prove A->False is true.)

```
Coq < Right.
1 subgoal
  A->False
  =====
  H : (A\ / A->False)->False
  A : Prop
```

```

Coq < Intro.
1 subgoal
  False
  =====
  H0 : A
  H : (A\A->False)->False
  A : Prop

```

(We again apply H)

```

Coq < Apply H.
1 subgoal
  A\A->False
  =====
  H0 : A
  H : (A\A->False)->False
  A : Prop

```

(This time we indicate with tactic Left that we want to prove that A is True. Since we already have the proof H0 of A, we use the Assumption tactic to indicate this)

```

Coq < Left;Assumption.
Goal proved!

```


(*****)

APPENDIX 2

Some Graph Problems in Coq

Program Development Through Proof Transformation

(*****)

Parameter V : Set.

Axiom eqV : (x,y : V){<V>x=y}+{~<V>x=y}.

Inductive Set list = nil : list | cons : V -> list -> list.

Local abs_prop =

[l:list](<Prop>Match l with
(* nil *) True
(* cons a m *) [a:V][m:list][P:Prop]False).

Goal (a:V)(m:list)~<list>nil=(cons a m).

Unfold not ; Intros a m h.

Change (abs_prop (cons a m)).

Elim h ; Simpl ; Auto.

Save nil_cons.

Goal (l : list){<list>nil=l}+{~<list>nil=l}.

Induction l.

Left;Auto.

Intros;Right;Apply nil_cons.

Save nil_or_not_nil.

Global app.

Body [l,m:list](<list>Match l with
(* nil *) m
(* cons a m *) [a:V][m:list](cons a))
: list->list->list.

(*list membership function*)

Definition occurs_in.

Body [l : list][x : V](<Prop>Match l with
False
[a : V][m : list][P : Prop]
<V>x=a) \ / P).

Goal (x : V)~(occurs_in nil x).

Intro;Simpl;Unfold not;Auto.

Save not_occurs_in_nil.

Goal (x : V)(v : V)(y : list)(~<V>x=v)->
(occurs_in (cons v y) x)->(occurs_in y x).

Simpl;Intros.

Elim H0.

Intro;Elim H;Assumption.

Intro;Assumption.
Save occurs_in_prop.

Goal (x : V)(l : list){(occurs_in l x)}+{~(occurs_in l x)}.
Intros;Elim l.
Right;Apply not_occurs_in_nil;Assumption.
Intros;Elim (eqV x v).
Intro;Left;Simpl;Left;Auto.
Intro;Elim H;Intros.
Left;Simpl;Right;Auto.
Right;Unfold not;Intro;Elim b0;
Apply occurs_in_prop with v;Auto.
Save occurs_in_or_not.

Goal (x : V)(la,lb:list)((occurs_in la x)\/(occurs_in lb x))->
 (occurs_in (app la lb) x).
Intros.
Elim H;Elim la;Simpl;Auto.
Intro Q;Elim Q.
Intros.
Elim H1;Auto.
Save occurs_in_app_propa.

Goal (x:V)(la,lb:list)(occurs_in (app la lb) x)->
 ((occurs_in la x)\/(occurs_in lb x)).
Intros until lb;Elim la;Simpl;Intros;Auto.
Elim H0;Auto.
Intros;Cut ((occurs_in y x)\/(occurs_in lb x));Intros.
Elim H2;Intros;Auto.
Auto.
Save occurs_in_app_propb.

(*remove vertice from list*)
Inductive Definition rm_V_list[x:V] : list->list->Prop =
 nil_rm_V_list : (rm_V_list x nil nil)
| cons_rm_V_list1 : (y : V)(l : list)(m : list)
 (rm_V_list x l m)->(~(<V>x=y))->
 (rm_V_list x (cons y l) (cons y m))
| cons_rm_V_list2 : (l : list)(m : list)
 (rm_V_list x l m)->
 (rm_V_list x (cons x l) m).

Goal (x : V)(l : list){lx : list | (rm_V_list x l lx)}.
Intros;Elim l.
Apply exist with nil;Apply nil_rm_V_list.
Intros;Elim (eqV x v);Intro;Elim H;Intros.
Apply exist with x0 ; Elim a.
Apply cons_rm_V_list2;Assumption.
Apply exist with (cons v x0).
Apply cons_rm_V_list1;Auto.
Save rm_V_list_fn.

(*property (a) a vertice does not occur in a list
 from which it is removed

```
property (b) vertices other than the removed vertice
remain in the list*)
```

```
Goal (l,lx : list)(x : V)(rm_V_list x l lx)->
  (~ (occurs in lx x)).
```

Intros.

Elim H.

Apply not occurs in nil;Auto.

Unfold * not; Intros.

Elim H1; Apply occurs in prop with y:Auto.

Auto.

Save rm V list propa.

```
Goal (x,y:V)(l,lx:list)(rm_V_list x l lx)->
  (~<V>x=y)->(occurs_in l y)->(occurs_in lx y).
```

Intros until 1x; Intro; Intro.

Elim H;Simpl;Intros;Auto.

Elim H4;Auto.

```
Elim H3;Auto;Intro;Elim H0;Auto.
```

Save rm V list propb.

Inductive Definition graf : Set =

```
nil graf : graf
```

```
1  cons_graf : V->list->graf->graf.
```

(*edges from a vertice in a graf*)

Inductive Definition edges from $\text{in}[v1:V] : \text{graf} \rightarrow \text{list} \rightarrow \text{Prop} =$

```
edges from nil : (edges from in vl nil graf nil)
```

```
edges from in l : (v2 : V)(l1 : graf)(t1 : list)(t2 : list)
```

```
(edges_from in vl ll tl)->(<V>vl = v2) ->
```

```
(edges from in v1 (cons graf v2 t2 ll) t2)
```

```
edges from in 2 : (v2 : V)(l1 : graf)(t1 : list)(t2 : list)
```

```
(edges_from_in v1 l1 t1) -> (~<V>v1 = v2) ->
```

```
(edges from in v1 (cons graf v2 t2 ll) t1).
```

```
Goal (v : V)(n : graf)
```

```
{m : list | (edges from in v n m)}.
```

Intros.

Elim n.

Apply exist with nil; Apply edges from nil.

Intros.

```
Elim (eqV v v0);Intros;Elim H;Intros.
```

Apply exist with l;Apply edges from in l with x ;Auto.

Apply exist with x; Apply edges from in 2; Auto.

Save edges from in fn.

Definition locala.

```
Body [G:graf][l,m:list](⟨Prop⟩Match G with
```

$$(\langle \text{list} \rangle l = m)$$

```
[a:V][b:list][c:graf][d:Prop]
```

True).

Definition localb.

```
Body [x:V][G:graf][l:list](<Prop>Match G with
```

```

True
[a:V][b:list][c:graf][d:Prop]
((<V>x=a)->(<list>b=l))).

```

Definition localc.

```

Body [x:V][G:graf][m:list](<Prop>Match G with
True
[a:V][b:list][c:graf][d:Prop]
((~<V>x=a)->(edges_from_in x c m))).

```

(*edge_list of a vertice in a graf is unique*)

```

Goal (G : graf)(x : V)(l,m : list)
(edges_from_in x G l)->(edges_from_in x G m)->(<list>l=m).
Induction G.
Intros;Change (locala nil_graf l m).
Elim H.
Elim H0;Simpl;Auto.
Simpl;Auto.
Simpl;Auto.
Intros until m;Elim (eqV x v);Intro.
Elim a;Intros;
Cut (GG:graf)(vert:V)(la,lb:list)
(edges_from_in vert (cons_graf vert la GG) lb)->
(<list>la=lb);Intros.
ElimType <list>l=m.
ElimType <list>l=l0;Auto.
Apply H2 with y x;Auto.
Apply H2 with y x;Auto.
Cut (<V>vert=vert);Auto.
Change (localb vert (cons_graf vert la GG) lb).
Elim H2;Simpl;Auto.
Intros.
Elim H5;Auto.
Cut (la:list)(edges_from_in x (cons_graf v l y) la)->
(edges_from_in x y la);Intros.
Cut (edges_from_in x y m).
Cut (edges_from_in x y l0);Intros.
Apply H with x;Auto.
Apply H0;Auto.
Apply H0;Auto.
Cut (~<V>x=v);Auto.
Change (localc x (cons_graf v l y) la);Elim H0;
Simpl;Auto.
Intros.
Elim H4;Auto.
Save edges_from_in_prop.

```

Inductive Definition edge[G:graf;x,y:V] : Prop =
edge_intro : (l : list)(edges_from_in x G l)->
(occurs_in l y)->(edge G x y).

```

Goal (x,y:V)~(edge nil_graf x y).
Unfold * not;Intros.
Elim H.

```

```
Intro;Intro;ElimType <list>nil=1.
Apply not_occurs_in_nil;Auto.
Cut (edges_from_in x nil_graf nil);Intros.
Apply edges_from_in_prop with nil_graf x;Auto.
Apply edges_from_nil.
Save edge_propa.
```

```
Goal (G:graf)(x,y:V)(l:list)(edge G x y)->
  (edges_from_in x G l)->(occurs_in l y).
Intros;Elim H;Intros.
ElimType(<list>l0=1);Auto.
Apply edges_from_in_prop with G x;Auto.
Save edge_propb.
```

```
Definition local.
Body [x:V][G:graf][m:list](<Prop>Match G with
  True
  [a:V][b:list][c:graf][d:Prop]
  ((~<V>x=a)->(edges_from_in x c m))).
```

```
Goal (G:graf)(x,y,z:V)(l:list)
  (edge (cons_graf x l G) y z)->(~<V>x=y)->(edge G y z).
Intros;Elim H;Intros.
Cut (edges_from_in y G l0);Intros.
Apply edge_intro with l0;Auto.
Cut (~<V>y=x).
Change (localc y (cons_graf x l G) l0);Elim H1;Simpl;Auto.
Intros.
Elim H6;Auto.
Unfold not;Intro;Elim H0;Auto.
Save edge_propc.
```

```
Goal (G:graf)(x,y,z:V)(l:list)
  (edge G y z)->(~<V>x=y)->(edge (cons_graf x l G) y z).
Intros.
Elim H;Intros.
Cut (edges_from_in y (cons_graf x l G) l0);Intros.
Apply edge_intro with l0;Auto.
Apply edges_from_in_2;Auto.
Unfold not;Intro;Elim H0;Auto.
Save edge_propd.
```

```
Goal (G : graf)(x,y : V){(edge G x y)}+{~(edge G x y)}.
Intros;Elim (edges_from_in_fn x G);Intros.
Elim (occurs_in_or_not y x0);Intros.
Left;Apply edge_intro with x0;Auto.
Right; Unfold not;Intros.
Elim b.
Apply edge_propb with G x;Auto.
Save edge_prope.
```

(*add a new edge to a graf*)

```
Inductive Definition cons_edge_graf[G:graf;x,y:V] : graf->Prop =
  cons_edge_graf_intro : (l : list)(edges_from_in x G l)->
```

```
(cons_edge_graf G x y (cons_graf x (cons y l) G)).
```

```
Goal (G : graf)(x : V)(y : V)
{Gxy : graf l (cons_edge_graf G x y Gxy)}.
Intros;Elim (edges_from_in_fn x G);Intros.
Apply exist with (cons_graf x (cons y x0) G).
Apply cons_edge_graf_intro;Auto.
Save cons_edge_graf_fn.
```

```
Goal (G,Gxy:graf)(x,y:V)(cons_edge_graf G x y Gxy)->
(edge Gxy x y).
Intros.
Elim H.
Intros;
Cut (edges_from_in x (cons_graf x (cons y l) G) (cons y l));
Intros.
Cut (occurs_in (cons y l) y);Intros.
Apply edge_intro with (cons y l);Auto.
Simpl;Auto.
Apply edges_from_in_l with l;Auto.
Save cons_edge_graf_propa.
```

```
Goal (x,y:V)(G,Gxy:graf)(cons_edge_graf G x y Gxy)->
((p,q:V)(edge G p q)->(edge Gxy p q)).
Intros.
Elim H;Intros.
Elim H0;Intros.
Elim (eqV p x);Intros.
Elim a;Intros.
Cut (edges_from_in p (cons_graf p (cons y l) G) (cons y l));Intros.
Cut (occurs_in (cons y l) q);Intros.
Apply edge_intro with (cons y l);Auto.
Simpl;Right.
ElimType (<list>l0=l);Auto.
Apply edges_from_in_prop with G x;Auto.
Elim a;Auto.
Apply edges_from_in_l with l0;Auto.
Apply edge_propd;Auto.
Unfold not;Intros;Elim b;Auto.
Save cons_edge_graf_propb.
```

```
(*add a graf to a graf*)
Inductive Definition union_graf[G:graf] : graf->graf->Prop =
  nil_union_graf : (union_graf G nil_graf G)
| cons_union_graf : (x : V)(la : list)(lb : list)
  (Gb : graf)(Gc : graf)
  (union_graf G Gb Gc)->(edges_from_in x G la)->
  (union_graf G (cons_graf x lb Gb)
    (cons_graf x (app la lb) Gc)).
```

```
Goal (Ga,Gb : graf){Gc : graf l (union_graf Ga Gb Gc)}.
Induction Gb;Intros.
Apply exist with Ga;Apply nil_union_graf.
Elim H;Intros.
```

```

Elim (edges_from_in_fn v Ga);Intros.
Apply exist with (cons_graf v (app x0 l) x).
Apply cons_union_graf;Assumption.
Save union_graf_fn.

Goal (Ga,Gb,Gab : graf)(union_graf Ga Gb Gab)->
  ((x,y:V)((edge Ga x y)\/(edge Gb x y))->
    (edge Gab x y)).
Intros until Gab;Intro;Elim H.
Intros;Elim H0;Auto.
Intro;Cut ~(edge nil_graf x y);Intros.
Elim H2;Auto.
Apply edge_propa.
Intros until y.
Elim (eqV x x0);Intro.
Elim a;Intros.
Cut (occurs_in (app la lb) y).
Cut (edges_from_in x (cons_graf x (app la lb) Gc)(app la lb)).
Intros;Apply edge_intro with (app la lb);Auto.
Elim (edges_from_in_fn x Gc);Intros.
Apply edges_from_in_1 with x1;Auto.
Elim H3;Intros.
Cut (occurs_in la y);Intros.
Apply occurs_in_app_propa;Auto.
Apply edge_propb with Ga x;Auto.
Cut (occurs_in lb y);Intros.
Apply occurs_in_app_propa;Auto.
Cut (edges_from_in x (cons_graf x lb Gb0) lb);Intros.
Apply edge_propb with (cons_graf x lb Gb0) x;Auto.
Elim (edges_from_in_fn x Gb0);Intros.
Apply edges_from_in_1 with x1;Auto.
Intros;Cut ((edge Ga x0 y)\/(edge Gb0 x0 y));Intros.
Elim (H1 x0 y H4);Intros.
Cut (edges_from_in x0 (cons_graf x (app la lb) Gc) l);Intros.
Apply edge_intro with l;Auto.
Apply edges_from_in_2;Auto.
Unfold not;Intro;Elim b;Auto.
Elim H3;Auto.
Intro;Right;Apply edge_propc with x lb;Auto.
Save union_graf_propa.

Goal (Ga,Gb,Gab : graf)(union_graf Ga Gb Gab)->
  (x,y:V)(edge Gab x y)->((edge Ga x y)\/(edge Gb x y)).
Intros until Gab;Intro;Elim H;Auto.
Intros until y;Elim (eqV x0 x);Intro.
Elim a;Intros.
Cut (edges_from_in x0 (cons_graf x0 (app la lb) Gc) (app la lb));
Intros.
Cut ((occurs_in la y) /\ (occurs_in lb y));Intros.
Elim H5;Intros.
Left;Apply edge_intro with la;Auto.
ElimType (<V>x=x0);Auto.
Right;Apply edge_intro with lb;Auto.
Elim (edges_from_in_fn x0 Gb0);Intros.

```

```

Apply edges_from_in_1 with x1;Auto.
Apply occurs_in_app_propb.
Apply edge_propb with (cons_graf x0 (app 1a 1b) Gc) x0;Auto.
Elim (edges_from_in_fn x0 Gc);Intros.
Apply edges_from_in_1 with x1;Auto.
Intro;Cut (edge Gc x0 y);Intros.
Cut ((edge Ga x0 y) \ / (edge Gb0 x0 y));Auto;Intros.
Elim H5;Auto.
Intros;Right;Apply edge_propd;Auto.
Unfold not;Intro;Elim b;Auto.
Apply edge_propc with x (app 1a 1b);Auto.
Unfold not;Intros;Elim b;Auto.
Save union_graf_propb.

```

```

(*remove vertice from a graf*)
Inductive Definition rm_V_graf[x:V] : graf->graf->Prop =
  nil_rm_V_graf      : (rm_V_graf x nil_graf nil_graf)
| cons_rm_V_graf1    : (Ga : graf)(Gb : graf)
  (l : list)(rm_V_graf x Ga Gb)->
  (rm_V_graf x (cons_graf x l Ga) Gb)
| cons_rm_V_graf2    : (y : V)(Ga : graf)(Gb : graf)
  (1a : list)(1b : list)(rm_V_list x 1a 1b)->((~(<V>x=y))->
  (rm_V_graf x Ga Gb)->
  (rm_V_graf x (cons_graf y 1a Ga) (cons_graf y 1b Gb))).

```

```

Goal (x : V)(G : graf){ Gx : graf | (rm_V_graf x G Gx)}.
Induction G;Intros.
Apply exist with nil_graf;Apply nil_rm_V_graf.
Elim H;Intros.
Elim (eqV x v);Intro.
Elim a;Apply exist with x0;Apply cons_rm_V_graf1;Auto.
Elim (rm_V_list_fn x l);Intros.
Apply exist with (cons_graf v x1 x0);Apply cons_rm_V_graf2;
Auto.
Save rm_V_graf_fn.

```

(*If a vertice is removed from a graf,
there is no edge from or to that vertice*)

```

Goal (x,y:V)(G,Gx:graf)(rm_V_graf x G Gx)->
  ((~(edge Gx x y))/\(~(edge Gx y x))).
Intros x y G Gx Q;Elim Q.
Split;Apply edge_propa;Auto.
Auto.
Unfold * not;Intros.
Elim H2;Intros;Split.
Intro;Elim H3;Apply edge_propc with y0 1b;Auto.
Unfold not;Intro;Elim H0;Auto.
Elim (eqV y y0);Intro.
Elim a;Intro.
Cut (edges_from_in y (cons_graf y 1b Gb) 1b);Intros.
Cut (occurs_in 1b x).
Change ~(occurs_in 1b x);Apply rm_V_list_propa with 1a;Auto.
Apply edge_propb with (cons_graf y 1b Gb) y;Auto.

```



```

Elim (edges_from_in_fn y Gb);Intros.
Apply edges_from_in_1 with x0;Auto.
Intro;Cut (edge Gb y x);Auto.
Apply edge_propc with y0 lb;Auto.
Unfold not;Intro;Elim b;Auto.
Save rm_V_graf_prop.

```

```

Inductive Definition pair_dfs : Set =
  pair_dfs_intro : (ga,gb : graf)(l : list)pair_dfs.

```

```

Definition pd_grafa.
Body [d : pair_dfs][<graf>Match d with
  [ga : graf][gb : graf][l : list]ga].

```

```

Definition pd_grafb.
Body [d : pair_dfs][<graf>Match d with
  [ga : graf][gb : graf][l : list]gb].

```

```

Definition pd_list.
Body [d : pair_dfs][<list>Match d with
  [ga : graf][gb : graf][l : list]l].

```

(*depth first search*)

```

Inductive Definition dfs: graf->V->list->list->
  pair_dfs->Set =
  init_dfs : (gin,gout : graf)(vert : V)(Qlin : list)
    (rm_V_graf vert gin gout)->
    (dfs gin vert nil Qlin
      (pair_dfs_intro gout nil_graf (cons vert Qlin)))
  | cons_dfsa : (gin : graf)(verta,vertb : V)(Qlin : list)
    (l: list)(p : pair_dfs)(dfs gin verta l Qlin p)->
    (occurs_in (pd_list p) vertb)->
    (dfs gin verta (cons vertb l) Qlin p)
  | cons_dfsb : (gin,Qgab,Qgc : graf)(verta,vertb : V)(Qlin : list)
    (la,lb : list)(pa,pb : pair_dfs)(dfs gin verta la Qlin pa)->
    (~ (occurs_in (pd_list pa) vertb))->
    (edges_from_in vertb (pd_grafa pa) lb)->
    (dfs (pd_grafa pa) vertb lb (pd_list pa) pb)->
    (union_graf (pd_grafb pa) (pd_grafb pb) Qgab)->
    (cons_edge_graf Qgab verta vertb Qgc)->
    (dfs gin verta (cons vertb la) Qlin
      (pair_dfs_intro (pd_grafa pb) Qgc (pd_list pb))).

```

(* removal of minimum one vertice with nonnil
edge list gives a proper subgraf *)

```

Inductive Definition subgraf[G : graf] :graf->Prop =
  subgraf_introa : (Gs : graf)(x : V)(l : list)
    (edges_from_in x G l)->(~<list>nil=l)->
    (rm_V_graf x G Gs)->(subgraf G Gs)
  | subgraf_introb : (Gs,Gss : graf)(x : V)
    (subgraf G Gs)->(rm_V_graf x Gs Gss)->
    (subgraf G Gss).

```

```

Goal (ga,gb,gc : graf)(subgraf ga gb)->(subgraf gb gc)->
  (subgraf ga gc).
Intros;Elim H0;Intros.

```

Apply subgraf_introb with gb x;Auto.
 Apply subgraf_introb with Gs x;Auto.
 Save subgraf_prop.

Variable well_founded : (A:Set)(A→A→Prop)→Prop.

Axiom graf_wf:(A : Set)(f:A→graf)
 (well_founded A [a1,a2:A](subgraf (f a2)(f a1))).

Axiom well_founded_recursion : (A:Set)(R:A→A→Prop)
 (well_founded A R)→(P:A→Set)
 (a:A)→((b:A)→((c:A)(R c b)→(P c))→(P b)) →(P a).

Definition id_graf.

Body [a : graf]a : graf→graf.

Inductive Definition exist_sp[A:Set;P:A→Set;Q:A→Prop]: Set=
 exist_sp_intro : (x : A)(P x)→(Q x)→(exist_sp A P Q).

Goal (gin : graf)(x : V)(l : list)(Qlin : list)
 (edges_from_in x gin l)→(¬(l=nil)→)
 (exist_sp pair_dfs
 [p:pair_dfs](dfs gin x l Qlin p)
 [p:pair_dfs](subgraf gin (pd_grafa p))).

Intro gin;Pattern gin;

Apply (well_founded_recursion graf [x,y:graf](subgraf y x)).

Cut (well_founded graf

[a1,a2 : graf](subgraf (id_graf a2) (id_graf
 a1))).

Unfold * id_graf;Intro;Exact H.

Apply graf_wf.

Intros;Elim l;Intros.

Elim(rm_V_graf_fn x b);Intros.

Apply exist_sp_intro with

(pair_dfs_intro x0 nil_graf (cons x Qlin));Simpl.

Apply init_dfs;Auto.

Apply subgraf_introa with x l;Auto.

Elim H2;Intros. *

Elim (occurs_in_or_not v (pd_list x0));Intro.

Apply exist_sp_intro with x0;Auto.

Apply cons_dfsa;Auto.

Elim (edges_from_in_fn v (pd_grafa x0));Intros.

Elim (nil_or_not_nil x1);Intros.

Elim (rm_V_graf_fn v (pd_grafa x0));Intros.

Cut (subgraf b x2);Intros.

Cut (dfs (pd_grafa x0) v x1 (pd_list x0)

(pair_dfs_intro x2 nil_graf (cons v (pd_list x0))));

Intros.

Elim (union_graf_fn (pd_grafb x0)

(pd_grafb (pair_dfs_intro x2 nil_graf

(cons v (pd_list x0))));Intros.

Elim (cons_edge_graf_fn x3 x v);Intros.

Apply exist_sp_intro with

(pair_dfs_intro

```

(pd_grafa (pair_dfs_intro x2 nil_graf (cons v (pd_list x0))))
x4
(pd_list (pair_dfs_intro x2 nil_graf (cons v (pd_list x0))))).
Apply cons_dfsb with x3 x1 x0;Auto.
Simpl;Auto.
Elim a;Apply init_dfs;Auto.
Apply subgraf_introb with (pd_grafa x0) v;Auto.
Elim (H (pd_grafa x0) q v x1 (pd_list x0) p0 b1);Intros.
Cut (subgraf b (pd_grafa x2)).
Elim (union_graf_fn (pd_grafb x0) (pd_grafb x2));Intros.
Elim (cons_edge_graf_fn x3 x v);Intros.
Apply exist_sp_intro with
(pair_dfs_intro (pd_grafa x2) x4 (pd_list x2));Simpl;Auto.
Apply cons_dfsb with x3 x1 x0;Auto.
Apply subgraf_prop with (pd_grafa x0);Auto.
Save dfs_prop.

```

Inductive Definition $\text{dfs_spec}[G:\text{graf};x:V;p:\text{pair_dfs}] : \text{Prop} =$
 $\text{dfs_spec_intro} : (l : \text{list})(\text{edges_from_in } x \text{ } G \text{ } l) \rightarrow$
 $(\text{dfs } G \text{ } l \text{ nil } p) \rightarrow$
 $(\text{dfs_spec } G \text{ } p).$

```

Goal (G : graf)(x : V)
  {p : pair_dfs | (dfs_spec G x p)}.
Intros;Elim (edges_from_in_fn x G);Intro.
Elim (nil_or_not_nil x0);Intro.
Elim a;Intros.
Elim (rm_V_graf_fn x G);Intros.
Apply exist with (pair_dfs_intro x1 nil_graf (cons x nil)).
Apply dfs_spec_intro with nil;Auto.
Apply init_dfs;Auto.
Intro H;Elim (dfs_prop G x x0 nil H b );Intros.
Apply exist with x1;
Apply dfs_spec_intro with x0;Auto.
Save dfs_prog.

```

Inductive Definition $\text{min}[x,y : \text{nat}] : \text{nat} \rightarrow \text{Prop} =$
 $\text{min_le} : (le \ x \ y) \rightarrow (\text{min } x \ y \ x)$
 $\mid \text{min_gt} : (gt \ x \ y) \rightarrow (\text{min } x \ y \ y).$

```

Goal (x,y:nat){z:nat | (min x y z)}.
Intros;Elim (le_or_gt x y);Intro.
Apply exist with x;Apply min_le;Auto.
Apply exist with y;Apply min_gt;Auto.
Save min_fn.

```

```

Goal (n,m : nat){<(nat>n=m)}+{~<(nat>n=m)}.
Induction n.
Induction m.
Left;Auto.
Intros;Elim H.
Intros;Right.
Elim a; Auto.
Intros;Auto.

```

```

Intros; Elim m.
ElimType {<nat>0=(S y)}+{~<nat>0=(S y)};Auto.
Intros;Right;Unfold not;Intros.
Elim b;Elim H0;Auto.
Intros;Elim (H y0);Intro;Auto.
Right;Unfold not;Intros;Elim b;Auto.
Save eqnat.

```

```

Inductive Definition ll : Set =
  nil_ll : ll
| cons_ll : list -> ll -> ll.

```

```

Inductive Definition rm_l_l[l:list]:list->list->Prop =
  rm_l_l_nil : (rm_l_l l nil nil)
| rm_l_l_consa : (la,lb : list)(v : V)(rm_l_l l la lb)->
  (occurs_in l v)->(rm_l_l l (cons v la) lb)
| rm_l_l_consb : (la,lb : list)(v : V)(rm_l_l l la lb)->
  (~(occurs_in l v))->(rm_l_l l (cons v la) (cons v lb)).

```

```

(*lc is the list of vertices in lb that are not in la*)
Goal (la,lb : list){lc : list | (rm_l_l la lb lc)}.
Intros;Elim lb;Intros.
Apply exist with nil;Apply rm_l_l_nil;Auto.
Elim H;Intro;Elim (occurs_in_or_not v la);Intros.
Apply exist with x;Apply rm_l_l_consa;Auto.
Apply exist with (cons v x);Apply rm_l_l_consb;Auto.
Save rm_l_l_fn.

```

```

(*for each vertex defines the position at which
it occurs in a list*)

```

```

Inductive Definition depth[x:V]:list->nat->nat->Prop =
  depth_nil : (depth x nil 0 0)
| depth_consa : (l : list)(m,n : nat)(depth x l m n)->
  (depth x (cons x l) (S m) (S m))
| depth_consb : (l : list)(v : V)(m,n : nat)
  (depth x l m n)->(~<V>x=v)->
  (depth x (cons v l) (S m) n).

```

```

Inductive Definition depth_spec[x:V;l:list]:nat->Set=
  depth_spec_intro : (m,n : nat)(depth x l m n)->
  (depth_spec x l n).

```

```

Goal (x : V)(l : list){n : nat & (depth_spec x l n)}.
Induction l.
Apply existS with 0;Apply depth_spec_intro with 0;
Apply depth_nil.
Intros;Elim H;Intros.
Elim p;Elim (eqV x v);Intros.
Elim a;Apply existS with (S m);
Apply depth_spec_intro with (S m);
Apply depth_consa with n;Auto.
Apply existS with n;
Apply depth_spec_intro with (S m);
Apply depth_consb;Auto.

```

Save depth_spec_fn.

```
Inductive Definition depth_min[lis:list;def:nat;dfs_list:list]
      :list->nat->Prop =
  depth_min_nil : (depth_min lis def dfs_list nil def)
| depth_min_consa : (l : list)(v : V)(m,n,o:nat)
  (depth_min lis def dfs_list l m)->(occurs_in lis v)->
  (depth_spec v dfs_list n)->(min m n o)->
  (depth_min lis def dfs_list (cons v l) o)
| depth_min_consb : (l : list)(v : V)(m:nat)
  (depth_min lis def dfs_list l m)->(~(occurs_in lis v))->
  (depth_min lis def dfs_list (cons v l) m).
```

```
Goal (lis : list)(def : nat)(dfs_list : list)(l : list)
  {n : nat | (depth_min lis def dfs_list l n)}.
Intros;Elim l;Intros.
Apply exist with def;Apply depth_min_nil.
Elim H;Elim (occurs_in_or_not v lis);Intros.
Elim (depth_spec_fn v dfs_list);Intros.
Elim (min_fn x x0);Intros.
Apply exist with x1;Apply depth_min_consa with x x0;Auto.
Apply exist with x;Apply depth_min_consb;Auto.
Save depth_min_fn.
```

```
Inductive Definition pair_scc : Set =
  pair_scc_intro : nat->list->ll->pair_scc.
```

Definition ps_num.

```
Body [p : pair_scc](<nat>Match p with
  [a : nat][c : list][d : ll]a).
```

Definition ps_list.

```
Body [p : pair_scc](<list>Match p with
  [a : nat][c : list][d : ll]c).
```

Definition ps_ll.

```
Body [p : pair_scc](<ll>Match p with
  [a : nat][c : list][d : ll]d).
```

Inductive Definition dfs_scc[Gin:graf]

```
  : graf->V->list->list->pair_dfs->
  pair_scc->pair_scc->nat->Prop =
  init_dfs_scc : (gin,gout,Qg : graf)(vert : V)(Qlin : list)
    (ps : pair_scc) (rm_V_graf vert gin gout)->
    (dfs_scc Gin gin vert nil Qlin
      (pair_dfs_intro gout Qg (cons vert Qlin)) ps
      (pair_scc_intro (S(ps_num ps)) (cons vert (ps_list ps))
        (ps_ll ps)) (ps_num ps))
| cons_dfs_scc1 : (gin : graf)(verta,vertb : V)
  (Qlin:list)(l : list)(minnat,newmin : nat)
  (p:pair_dfs)(pin,pout : pair_scc)
  (dfs_scc Gin gin verta l Qlin p pin pout minnat)->
  (occurs_in (pd_list p) vertb)->
  (depth_min (ps_list pout) minnat (pd_list p)
    (cons vertb nil) newmin)->
  (dfs_scc Gin gin verta (cons vertb l)
```

```

    Qlin p pin pout newmin)
  I cons_dfs_scc2 : (gin,Qg:graf)(verta,vertb:V)
    (Qlin,la,lb,lc,lis:list)(pa,pb : pair_dfs)
    (psa,psb,psc : pair_scc)(minnata,minnatb,newmin : nat)
    (dfs_scc Gin gin verta la Qlin pa psa psb minnata)->
    (~{occurs_in (pd_list pa) vertb})->
    (edges_from_in vertb (pd_grafa pa) lb)->
    (dfs_scc Gin (pd_grafa pa) vertb lb (pd_list pa)
      pb psb psc minnatb)->
    (edges_from_in vertb Gin lc)->
    (depth_min (ps_list psb) minnatb (pd_list pb) lc newmin)->
    (<nat>(ps_num psb)=newmin)->
    (rm_l_1 (ps_list psb) (ps_list psc) lis)->
    (dfs_scc Gin gin verta (cons vertb la) Qlin
      (pair_dfs_intro (pd_grafa pb) Qg (pd_list pb))
      psa (pair_scc_intro (ps_num psc)
        (ps_list psb) (cons_l1 lis (ps_l1 psc))) minnata)
  I cons_dfs_scc3 : (gin,Qg:graf)(verta,vertb:V)
    (Qlin,la,lb,lc:list)(pa,pb : pair_dfs)
    (psa,psb,psc : pair_scc)(minnata,minnatb,newmin,nm : nat)
    (dfs_scc Gin gin verta la Qlin pa psa psb minnata)->
    (~{occurs_in (pd_list pa) vertb})->
    (edges_from_in vertb (pd_grafa pa) lb)->
    (dfs_scc Gin (pd_grafa pa) vertb lb (pd_list pa)
      pb psb psc minnatb)->
    (edges_from_in vertb Gin lc)->
    (depth_min (ps_list psb) minnatb (pd_list pb) lc newmin)->
    (~(<nat>(ps_num psb)=newmin))->
    (min minnata newmin nm)->
    (dfs_scc Gin gin verta (cons vertb la) Qlin
      (pair_dfs_intro (pd_grafa pb) Qg (pd_list pb))
      psa psc nm).

```

```

Goal (Gin,gin : graf)(x : V)(la : list)(Qlin : list)
  (p:pair_dfs)(dfs gin x la Qlin p)->(psa:pair_scc)
  {psb:pair_scc&{nm:nat|(dfs_scc Gin gin x la Qlin
    p psa psb nm)}}.

```

```

Intros Gin gin x la Qlin p H;Elim H;Intros.

```

```

Apply existS with (pair_scc_intro (S (ps_num psa))
  (cons vert (ps_list psa)) (ps_l1 psa));

```

```

Apply exist with (ps_num psa);

```

```

Apply init_dfs_scc;Auto.

```

```

Elim (H0 psa);Intros.

```

```

Elim p1;Intros.

```

```

Elim (depth_min_fn (ps_list x0) x1 (pd_list p0)
  (cons vertb nil));Intros.

```

```

Apply existS with x0; Apply exist with x2;

```

```

Apply cons_dfs_scc1 with x1;Auto.

```

```

Elim (H0 psa);Intros.

```

```

Elim p0;Intros.

```

```

Elim (H1 x0);Intros.

```

```

Elim p2;Intros.

```

```

Elim (edges_from_in_fn vertb Gin);Intros.

```

```

Elim (depth_min_fn (ps_list x0) x3 (pd_list pb) x4);Intros.

```

```

Elim (eqnat (ps_num x0) x5);Intros.
Elim (rm_l_l_fn (ps_list x0) (ps_list x2));Intros.
Apply existS with (pair_scc_intro (ps_num x2) (ps_list x0)
    (cons_ll x6 (ps_ll x2)));
Apply exist with x1;
Apply cons_dfs_scc2 with lb x4 pa x3 x5;Auto.
Elim (min_fn x1 x5);Intros.
Apply existS with x2;Apply exist with x6;
Apply cons_dfs_scc3 with lb x4 pa x0 x1 x3 x5;Auto.
Save dfs_scc_prop.

Inductive Definition dfs_scc_spec[g:graf;x:V;lla:ll] : Prop =
  dfs_scc_spec_intro : (p : pair_dfs)(ps : pair_scc)
    (mn : nat)(la : list)(edges_from_in x g la)->
    (dfs_scc g g x la nil p
      (pair_scc_intro (S 0) nil nil_ll) ps mn)->
    (<ll>lla = (cons_ll (ps_list ps) (ps_ll ps)))->
    (dfs_scc_spec g x lla).

Goal (G : graf)(x : V){lla : ll | (dfs_scc_spec G x lla)}.
Intros;Elim (edges_from_in_fn x G);Intro.
Elim (nil_or_not_nil x0);Intro.
Elim a;Intros.
Apply exist with (cons_ll (cons x nil) nil_ll).
Elim (rm_V_graf_fn x G);Intros.
Cut (dfs_scc G G x nil nil
  (pair_dfs_intro x1 nil_graf (cons x nil))
  (pair_scc_intro (S 0) nil nil_ll)
  (pair_scc_intro
    (S (ps_num (pair_scc_intro (S 0) nil nil_ll)))
    (cons x (ps_list (pair_scc_intro (S 0) nil nil_ll)))
    (ps_ll (pair_scc_intro (S 0) nil nil_ll)))
    (ps_num (pair_scc_intro (S 0) nil nil_ll))).
Intro H;Apply dfs_scc_spec_intro with
  (pair_dfs_intro x1 nil_graf (cons x nil))
  (pair_scc_intro
    (S (ps_num (pair_scc_intro (S 0) nil nil_ll)))
    (cons x (ps_list (pair_scc_intro (S 0) nil nil_ll)))
    (ps_ll (pair_scc_intro (S 0) nil nil_ll)))
    (ps_num (pair_scc_intro (S 0) nil nil_ll))) nil;Auto.
Apply init_dfs_scc;Auto.
Intro H;Elim (dfs_prop G x x0 nil H b);Intros.
Elim (dfs_scc_prop G G x x0 nil x1 p
  (pair_scc_intro (S 0) nil nil_ll));Intros.
Elim p0;Intros.
Apply exist with (cons_ll (ps_list x2) (ps_ll x2)).
Apply dfs_scc_spec_intro with x1 x2 x3 x0;Auto.
Save dfs_scc_prog.

```

REFERENCES

1. A.V.Aho, J.E.Hopcroft, and J.D.Ullman, "The Design and Analysis of Computer Algorithms", Addison Wesley[1990]
2. R.C.Backhouse. "Constructive Type Theory: A perspective from Computing Science." in Formal Development of Programs and Proofs, Ed. E.W.Dijkstra, Addison Wesley[1990].
3. H.Barendregt and K.Hemerik. "Types in Lambda Calculi and Programming Languages", ESOP'90, LNCS 432[1990].
4. T.Coquand. "On the Analogy between Propositions and Types" in Logical Foundations of Functional Programming, Ed. G.Huet, Addison Wesley[1990].
5. J.Y.Girard, P.Taylor, and Y.Lafont. "Proofs and Types." Cambridge University Press[1989].
6. G.Huet. "A Uniform Approach to Type Theory." in Logical Foundations of Functional Programming, Ed. G.Huet, Addison Wesley[1990].
7. G.Huet. "The Gilbreath Trick: A case study in Axiomatisation and Proof Development in the Coq Proof Assistant.". INRIA, [June 1991].
8. M.L.Minsky. "Computation: Finite and Infinite Machines", Prentice Hall[1967].

9. B.Pierce, S.Dietzen, and S.Michaylov. "Programming in Higher Order Typed Lambda Calculi", CMU Technical Report, CMU-CS-89-111[1989].
10. F.Pfenning, and Ch. Paulin-Mohring. "Inductively Defined Types in the Calculus of Constructions", MFPLS'89, LNCS[1989].
11. The CoC Documentation and User's Guide, INRIA[1989].
12. The Coq Proof Assistant User's Guide, INRIA[1991].